# 

**50.004**

**2D 2-SAT ALGORITHM ANALYSIS**

**Group 19**

# 

# Members:

Tan Xin Yi 1004226

Lim Jun Wei 1004379

Kiran Mohan 1004436

David Lim 1004485

Goh Yi Ern 1004266

Pan Zishan 1004564

# Table of Contents

[**50.004**](#_yci9p6eg6v9p)1

[**2D 2-SAT ALGORITHM ANALYSIS**](#_2r2tsnt5wdn2)1

[**Members:**](#_fqkt1d7mlnb9)1

[**Table of Contents**](#_sh5h3nhx2gii)2

[**Abstract**](#_n3lkma820x5g)3

[**Designing the algorithm**](#_k5o562y734sn)3

[**Implication Graph**](#_w2v0fq432sh8)3

[**Strongly Connected Components (SCCs)**](#_lckq6d6yggv1)4

[**Kosaraju’s Algorithm**](#_ow2kn2l2ixa5)4

[**DFS on the Graph**](#_g27e3f88ihll)4

[**Transpose of the Graph**](#_46ua2vu5ylk7)5

[**DFS on the Transposed Graph**](#_bz8izibo6pka)5

[**Putting it all together**](#_ql9yehgf88hj)5

[**How we use the implication graph to determine satisfiability**](#_61o9ic8n8jy0)6

[**2-SAT vs 3-SAT**](#_dbo5frem1kq)6

[**Implementation**](#_23hmc5uvbl6n)6

[**Class dir\_graph**](#_fox38i6fjnm)7

[**Other Helper Functions**](#_p7flh64fejgi)7

[**Putting It All Together**](#_8prn8xxq5tmq)8

[**Performance Analysis**](#_u1x8ao7t3iat)8

[**Time Complexities**](#_g1aenos4pd4z)8

[**Overall Running Time**](#_vdee0qwgs2w7)9

[**Testing**](#_hhg9zokjbqj)10

[**Conclusion**](#_99oxlzcf630p)10

[**Bonus: Randomizing Algorithm vs Deterministic Algorithm**](#_vjn3z1vzvl0l)11

# 

# Abstract

The SAT problem was the first problem to be proven as NP-complete, meaning that there is no algorithm that can efficiently solve all SAT problems. However, the 2-SAT problem, a specific subset of the SAT problem, has been shown to be solvable in polynomial time. Hence, in this report, our team aims to design an algorithm to solve the 2-SAT problem in polynomial time. Using the approach of creating an implication graph and identifying strongly connected components using Kosaraju’s Algorithm, our team is able to determine whether a given 2-SAT problem is satisfiable in polynomial time. This algorithm was then implemented in Python, and its performance and time complexity were analyzed. At the end of our report, we compare this algorithm with the randomizing algorithm to determine whether the randomizing algorithm can be a practical substitute for our initial deterministic algorithm.

# Designing the algorithm

## Implication Graph

A 2-SAT problem limits the problem of SAT to only those boolean formulas where each clause of the CNF is only made of 2 terms, also known as the 2-CNF.

One example of a 2-SAT problem could be:

For the value of the CNF to be true, each and every clause in the CNF has to be true as the CNF is a product of sums. Any particular clause of the CNF say can be further split into 2 clauses.

when

* If must be 1 i.e.)
* If must be 1 i.e.)

Thus,

is equivalent to

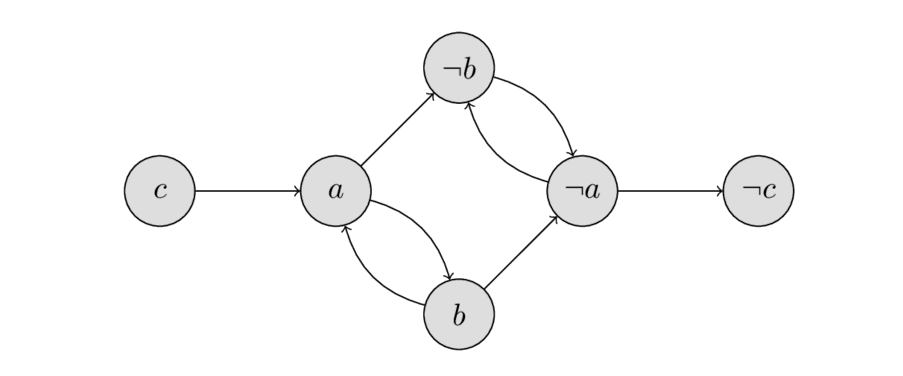
Hence, we can now express the CNF as implications and therefore make an implication graph with 2 edges for each clause based on the equivalent formula above.

is expressed in the implication graph as and

Note:

* As there are 2 edges for each clause, the total number of edges is 2m
* The total number of nodes is the number of boolean variables involved in the formula

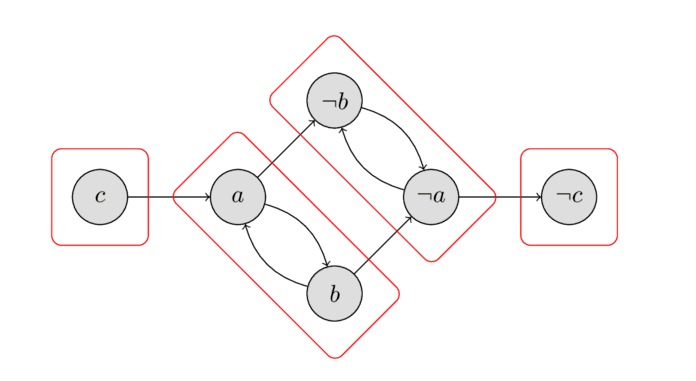
For example, a 2-SAT of the form would have the following edges:



*Src: https://cp-algorithms.com/graph/2SAT.html*

### Strongly Connected Components (SCCs)

A directed graph is said to be strongly connected if each vertex is reachable from every other vertex. A strongly connected component of a directed graph is a maximal strongly connected subgraph. The SCC’s for the above graph are circled in red below:



*Source: https://cp-algorithms.com/graph/2SAT.html*

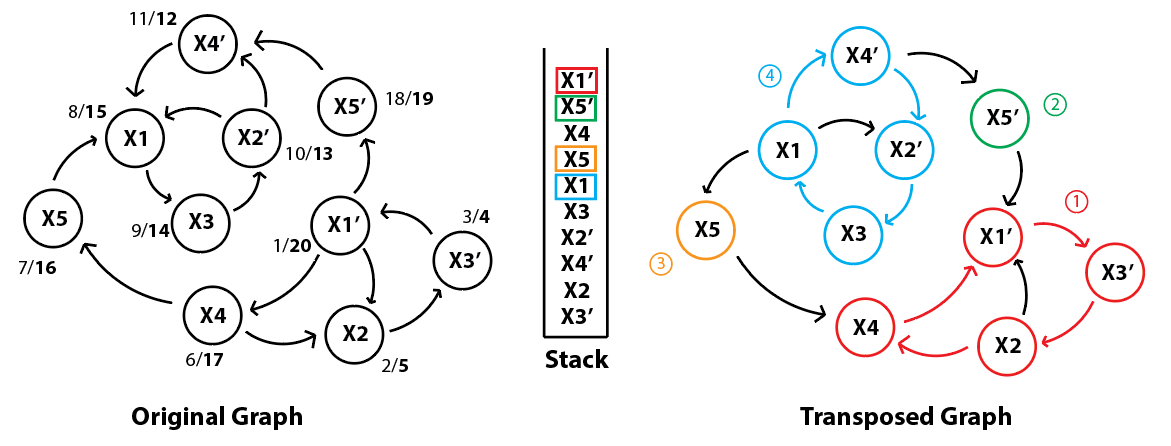
## Kosaraju’s Algorithm

Kosaraju’s algorithm is used to determine the strongly connected components of a graph. The Algorithms is divided into 3 parts, namely:

* Depth First Search (DFS) on the original graph
* Transpose of the graph
* Depth First Search (DFS) on the transposed graph

### DFS on the Graph

Perform a Depth First Search on the original graph whilst keeping track of the finish times of each node. This can be done using a stack. When a DFS finishes, put the source vertex on the stack. This way, nodes with the highest finishing time will be on top of the stack.

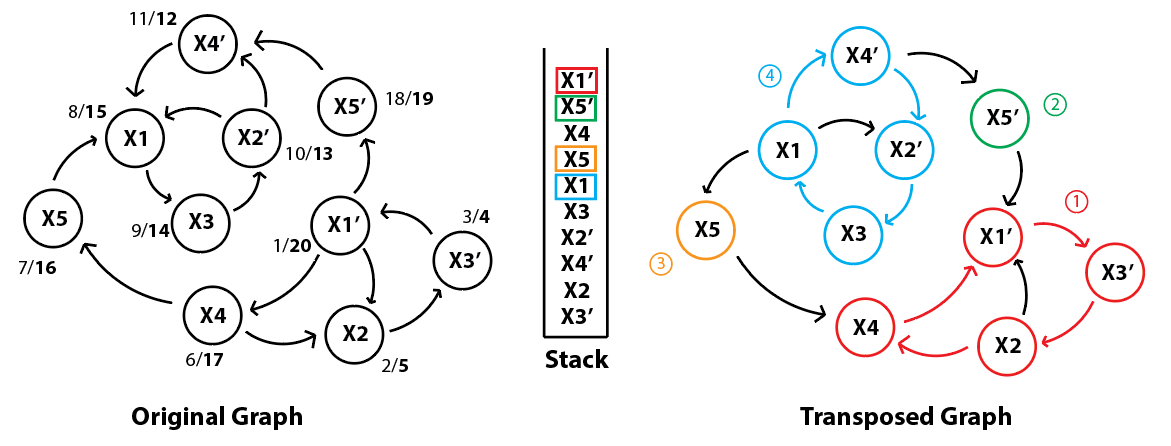


### Transpose of the Graph

Reverse the direction of all the arcs in order to get the transpose of the graph. This can be done using the adjacency list.

### DFS on the Transposed Graph

Perform a Depth First Search on the reversed graph, with the source vertex as the vertex on top of the stack. When the DFS finishes, all nodes visited will form one Strongly Connected Component. If any more nodes remain unvisited, this means there are more Strongly Connected Components, so pop vertices from top of the stack until a valid unvisited node is found. This step is repeated until all nodes are visited.



## Putting it all together

The Implication Graph and the Strongly Connected Components (SCC) play a huge role in the 2-SAT solver allowing the result to be computed in polynomial time. After converting the CNF into the graph, we run the Kosaraju algorithm on the graph in order to find the Strongly Connected Components of the graph.

We then go through each of the Strongly Connected Components of the graph in order to check if both a node, say , and its complement, , are found in the same. If so, the CNF is UNSAT.

## How we use the implication graph to determine satisfiability

Case 1: Suppose edge exist within a SCC, a possible solution for a would be a = true.

Case 2: Suppose edge exist within a SCC, a possible solution for a would be a = false.

Case 3: However, if both edges and exists within the same SCC, there would be a contradiction as both a = true and a = false will result in the other edge being false, which means that the CNF is unsatisfiable.

Hence, as long as we can prove that no such contradictions exist within the SCCs generated, the CNF formula is satisfiable.

## 2-SAT vs 3-SAT

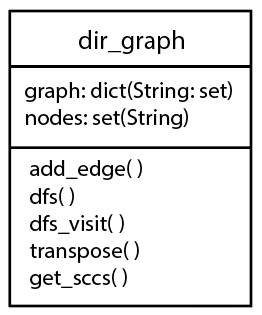
As previously mentioned, in a 2-SAT problem, since each clause contains only 2 literals, and all clauses must be true for the entire formula to be satisfiable, we can conclude that if one literal in a clause is false, then the other must be true.

For a 3-SAT problem, we have 3 literals in each clause. Given the clause , if is false, it only implies that either or is true. We are unable to definitively tell whether only is true, only is true or both and are true. Since the implication graph must be constructed with a definite value of each literal derived from the implications, implications drawn from a 3-SAT clause are insufficient for creating an appropriate implication graph. As a result, our chosen algorithm of creating an implication graph and identifying strongly connected components to determine satisfiability will not work for 3-SAT problems, as well as for any other N-SAT problems, where N > 2.

# Implementation

Our team has implemented our algorithm for the 2-SAT solver in Python. The code can be found in the file 2sat.py.

## Class dir\_graph



*Class diagram of dir\_graph*

In our implementation, we created the class *dir\_graph* to serve as our directed implication graph. This class contains the attributes *graph* and *nodes*, as well as the methods *add\_edge*, *dfs*, *dfs\_visit*, *transpose* and *get\_sccs*. The methods *dfs*, *dfs\_visit* and *transpose* can be considered as helper methods for the *get\_sccs* method as they will only be used internally in *get\_sccs*.

The *graph* attribute is a dictionary with its keys being the nodes of the graph and its values being the set of all nodes that that node directs to. The *nodes* attribute contains the set of all nodes in the graph.

The method *add\_edge* adds an edge to the graph, as well as new nodes, if the literals being added are not already present in the graph. *dfs* loops through all nodes of the graph and calls the recursive *dfs\_visit* method to conduct a depth-first search, in which the stack for Kosaraju’s Algorithm is created and the strongly connected components are identified. By adding the node to the stack after making the recursive calls, we allow the node with the earliest finish time to be added to the stack first. The conditional check for whether the current node has been visited also works to indicate whether the current node is included in the strongly connected component.

*transpose* returns a copy of our original implication graph with all edge directions reversed. *get\_sccs* implements Kosaraju’s Algorithm by first obtaining the stack of literals from the original implication graph by conducting a depth-first search of the entire graph, then iteratively popping off the topmost node of the stack and calling *dfs\_visit* while using that popped node as the source node to conduct depth-first searches on the transposed graph to identify all strongly connected components of the graph. A list of all strongly connected components is returned.

## Other Helper Functions

Our implementation also includes a number of helper functions outside of the *dir\_graph* class to aid us in determining the satisfiability of the formula and obtaining the values of each variable if the formula is satisfiable. These functions include *parse\_cnf*, *double\_neg*, *contradicts* and *get\_values*.

*parse\_cnf* helps to parse a .cnf file and returns the lists of all variables and all clauses. *double\_neg* is a convenience function for formatting literals appropriately when double negatives are present (ie. rewriting --3 into 3). *contradicts* checks for whether there are any instances of a variable and its negation appearing in a strongly connected component and returns true if there are. *get\_values* returns a list of the appropriate values of each variable given that the entire formula is satisfiable.

## Putting It All Together

The above-mentioned class methods and helper functions are put together in the *main* function of the 2sat.py file as the *main* function takes care of the general logic of the 2-SAT solver, from reading from the .cnf file to generating the final satisfiability result.

In *main*, we first call *parse\_cnf* to obtain both the list of variables and the formula, before creating an instance of the *dir\_graph* class and looping through the list of clauses in the formula to fill up the implication graph using the *dir\_graph.add\_edge* method. Then, we obtain the list of all strongly connected components using the *dir\_graph.get\_sccs* method, and check for contradictions using *contradicts*. If *contradicts* returns true, we output that the formula is unsatisfiable. Otherwise, we output that the formula is satisfiable and use *get\_values* to obtain the appropriate values of all variables.

# 

# Performance Analysis

## Time Complexities

Let the cnf input contain V literals. As mentioned earlier, each clause will have 2 implications as the limitation of a 2-SAT problem. The cnf will be converted into a graph with V nodes and E edges. Breakdown of each function’s running time:

*add\_edge* : O(1)

*double\_neg* : O(1)

*dfs* : We would iterate through each node V and call *dfs\_visit()* if the node has not been iterated yet. *dfs\_visit()* would recursively iterate through each node’s neighbours. After looping through each node, *dfs\_visit()* would have been called 2E times. Time complexity is Θ(V + E)

*transpose*: As we iterate through each node V, we iterate through its neighbours, using *add\_edge()* to reverse the direction such that the neighbour now points to the node. *add\_edge()* would effectively be called V+E times and since time complexity of *add\_edge()* is O(1), time complexity for this function is Θ(V + E)

*get\_sccs*: We first populate our stack using *dfs()* and then create a transposed version of the graph using *transpose()*. Our stack would have a length of V and for each node in the stack, we would use *dfs\_visit()* on the transposed graph to populate a list (sccs) of strongly connected components(scc). Similarly like before, *dfs\_visit()* would be called 2E times as we iterate through each neighbour of each node. So time complexity of the while loop in get\_scss is also Θ(V + E) like *dfs()* and *transpose()*. Thus the overall time complexity of *get\_scss()* is Θ(V + E) as well.

*(Note: In the worst case where the graph is complete, E = VP2 = V! / (V-2)! = V\*(V-1) in a directed graph, so time complexity would be O(V^2) for any function with time complexity Θ(V + E))*

*contradicts*: The list that contains all sublists of strongly connected components(scc), sccs, will always contain every node of the graph = V. The maximum number of scc sccs can store is V and this occurs when each node is an scc by itself. When every node in the graph forms one scc, sccs will only contain 1 scc populated by V nodes. Effectively, this means that the number of iterations for the double for loop in *contradicts()* is at most V. Thus the time complexity of this function is O(V) when there are no contradictions found.

*get\_values*: Using the explanation for *contradicts()*, this function will have a time complexity of O(V) as it will iterate through every item in sccs using a double for loop.

*parse\_cnf*: To determine the maximum number of clauses for a 2-SAT cnf with V literals (both x and !x are 2 literal), we can use VP2. Since VP2 = V\*(V-1), the worst case running time of this function is O(V^2) as the function iterates through each clause. The number of clauses is equal to E, the number of edges of the graph. *(This can be left out of the computation of the time complexity for the SAT solver as it is not part of the SAT solver algorithm)*

## Overall Running Time

To analyse the full running time, we will look at *main().* First it calls *parse\_cnf()* which returns an array of the clauses which is used to convert the boolean equation into a graph(V,E) using *add\_edges()* for each clause. Following this, the function calls *get\_sccs()* to return a list of all the strongly connected components, sccs, which has a time complexity of O(V+E). It then uses *contradicts()* that runs in O(V) to check for any errors and if no contradictions are found, it finally calls *get\_values()* which also runs in O(V) to return the literal values which will make the boolean equation satisfiable, representing the worst case running time scenario. Thus comparing the time complexity of each function *main()* calls, we can say that the overall worst case time complexity for this function is O(V+E) and our algorithm runs in polynomial time.

## Testing

To test our algorithm, our team used the CNF from the previous subsection on Kosaraju’s Algorithm. The following table shows the time elapsed from both opening the .cnf file and after parsing the file content to printing the satisfiability result.

|  |  |  |
| --- | --- | --- |
| **Attempt Number** | **Running Time from Opening File (ms)** | **Running Time from After Parsing File Content (ms)** |
| 1 | 1.32465 | 1.32465 |
| 2 | 1.99628 | 0.99850 |
| 3 | 3.99065 | 1.99413 |
| 4 | 1.99413 | 0.99659 |
| 5 | 1.99485 | 0.99730 |
| 6 | 1.99509 | 0.99802 |
| 7 | 2.39635 | 1.39856 |
| 8 | 4.25124 | 2.95162 |
| 9 | 0.99540 | 0.99540 |
| 10 | 1.99485 | 0.99683 |

From the ten recorded attempts, the average time taken from after parsing the file content to printing the satisfiability result is **1.36516ms** while the average time taken from opening the .cnf file to printing the result is **2.29335ms**.

# Conclusion

In this report, we identified that a boolean equation limited to 2 literals per clause, a 2-SAT, can be solved in polynomial time. The boolean equation is converted into a graph via implications and it is identified that if the nodes are part of the same strongly connected component, the equation is unsatisfiable. Our algorithm for a 2-SAT solver implements Kosaraju's algorithm to identify the strongly connected components. By taking the time complexity of the various functions in the algorithm into account, we determined the running time to be O(V+E). This algorithm strictly works for 2-SAT problems.

# 

# Bonus: Randomizing Algorithm vs Deterministic Algorithm

The randomizing algorithm to check for satisfiability for our group was implemented as such:

1. Initialize all variables to false
2. Loop through all clauses and check if all clauses are satisfied. If unsatisfiable clause is found:
3. Randomly select one of the two variables and, through another randomized selection, set that variable to either true or false
4. Repeat steps 2 and 3 until either all clauses are satisfied or stop after a set number of steps and conclude that there is no solution. The number of steps is the maximum number of times that we will loop through all clauses and can be set to any positive integer.

Since the randomizing algorithm works based on the arbitrary assignment of Boolean values to random variables of unsatisfiable clauses, there is a probability that this algorithm is unable to find a solution even though a solution might exist. This probability can be reduced by increasing the number of steps mentioned above as this gives the program more opportunities to re-assign Boolean values to variables, thus increasing the likelihood of finding a solution whereby all clauses are satisfiable. By Markov's inequality, the probability to not have found a solution after 100steps is at most .

To determine whether the randomizing algorithm is a practical substitute for the deterministic one, we tested the randomizing algorithm with the same .cnf file used to test the deterministic one above. For our tests, we varied the number of steps and, for each number of steps, we ran the program 10 times and recorded both the accuracy of the output and the average time taken from after parsing the .cnf file to printing the satisfiability result. Since the .cnf file is of a satisfiable CNF, we add 1 point to the accuracy score if the final result shows satisfiable.

|  |  |  |
| --- | --- | --- |
| **Number of Steps** | **Accuracy** | **Average Time Taken (ms)** |
| 10 | 2/10 | 1.36984 |
| 50 | 7/10 | 1.54465 |
| 100 | 10/10 | 7.54263 |
| 2500 (100) | 10/10 | 25.71912 |

Based on the results from the table above, we can conclude that the randomizing algorithm is reliable only for sufficiently large numbers of steps of approximately 100 steps or higher. However, we can also see that the average time taken for the randomization algorithm increases as the number of steps increases, and for larger steps, is greater than that of the deterministic one.

Taking into account that the variance of the time taken for each individual run of the program is relatively large, our team believes that we can find a balance point at approximately 100 steps whereby the accuracy is sufficiently high and the time taken is not too long, and at this point, the randomizing algorithm can be a practical substitute for the deterministic algorithm.